

# 1 Lexer

The Lexer reads the proof code and produces the list of tokens that represents it.

---

```
module Lexer where
  import Data.Char
  import Data.List.Split
```

---

These tokens are represented as `LexerTokens`, and are as follows

---

```
data Token = BOF
           | ID String
           | Natural Int
           | LParen
           | RParen
           | LBrack
           | RBrack
           | SQuote
           | DQuote
           | Arrow
           | Lambda
           | Exists
           | ForAll
           | OpAdd
           | OpSub
           | OpMul
           | OpDiv
           | OpMod
           | OpEqual
           | OpLT
           | OpGT
           | OpAnd
           | OpOr
           | Negation
           | Bottom
           | Comma
           | Colon
           | Equiv
           | Type
           | True
           | False
           | Null
           | Undefined
```

```

| TBoolean
| TNatural
| TSymbol
| TList
| TChar
| TypeOf
| Let
| Native
| EOF
deriving (Show)

```

---

The string itself is chopped up by the `munch` function, which uses the simplified maximal munch algorithm to produce tokens.

---

```

munch :: String → (Token, String)
munch = extractTokenStr Start ""

```

---

As you may have noticed, the actual munching is performed by `extractTokenStr`, while `munch` simply serves as an entry point. Before defining that, however, we require a few helper definitions.

First are the states which the state machine used for munching can be in:

---

```

data State = Start | Identifier | Number | Single | PossibleArrow |
            PossibleEquiv

```

---

Then, we have a few helper functions which can identify classes of characters. `isIdent` checks that a character is a valid character for an identifier, i.e. alphanumeric, or an underscore.

`isSingle` checks that a character is one of the characters that makes up a whole token on its own.

---

```

isIdent :: Char → Bool
isIdent c = isAlphaNum c || c == '_'

isSingle :: Char → Bool
isSingle c = c `elem` "()[]<>+-,=/%*/: |&'\"

extractTokenStr :: State → String → String → (Token, String)

```

```

extractTokenStr state token code = case state of
  Start      → case code of
    '-' : rest      → extractTokenStr PossibleArrow "-" rest
    ':' : rest      → extractTokenStr PossibleEquiv ":" rest
    l : rest | isAlpha l | l == '_'
                    → extractTokenStr Identifier [l] rest
    n : rest | isDigit n → extractTokenStr Number [n] rest
    o : rest | isSingle o → extractTokenStr Single [o] rest
    w : rest | isSpace w → extractTokenStr Start [] rest
    -      → error $ "Lexer could not process character
        sequence " ++ code -- TODO: LexerError
  Identifier → case code of
    l : rest | isIdent l → extractTokenStr Identifier (l : token)
    rest                → (convertToToken token, code)
    -                  → (convertToToken token, code)
  Number     → case code of
    l : rest | isDigit l → extractTokenStr Number (l : token) rest
    -                  → (Natural $ read $ reverse token, code)
  PossibleArrow → case code of
    '>' : rest      → extractTokenStr Single ">" rest
    -      → (convertToToken token, code)
  PossibleEquiv → case code of
    '=' : rest      → extractTokenStr Single "=" rest
    -      → (convertToToken token, code)
  Single      → (convertToToken token, code)

```

---

The strings extracted by `extractTokenStr`, other than the numeric ones, are converted to actual tokens by `convertToToken`. This function expects that the token be written backwards because that's how `extractTokenStr` makes them.

Is that a stupid design for this function? Probably, but I think it will be ok.

---

```

convertToToken :: String → Token
convertToToken ""      = Arrow
convertToToken ">"      = Arrow
convertToToken "("      = LParen
convertToToken ")"      = RParen
convertToToken "["      = LBrack
convertToToken "]"      = RBrack
convertToToken "<"      = OpLT
convertToToken ">"      = OpGT
convertToToken "-"      = OpSub
convertToToken "+"      = OpAdd
convertToToken "'"      = SQuote
convertToToken "\""     = DQuote
convertToToken "/"      = OpDiv

```

```

convertToToken "*"      = OpMul
convertToToken "="      = OpEqual
convertToToken "%"      = OpMod
convertToToken ":"      = Colon
convertToToken ","      = Comma
convertToToken ""       = Exists
convertToToken ""       = Lambda
convertToToken "\\\"     = Lambda
convertToToken "stsize" = Exists
convertToToken ""       = ForAll
convertToToken "llarof" = ForAll
convertToToken " "      = Negation
convertToToken ""       = Bottom
convertToToken "dna"    = OpAnd
convertToToken "&"      = OpAnd
convertToToken ""       = OpAnd
convertToToken "ro"     = OpOr
convertToToken "|"      = OpOr
convertToToken ""       = OpOr
convertToToken "=: "    = Equiv
convertToToken ""       = Equiv
convertToToken ""       = Native
convertToToken "llun"   = Null
convertToToken "denifednu" = Undefined
convertToToken "epyT"   = Type
convertToToken "foeptyt" = TypeOf
convertToToken "tel"    = Let
convertToToken "eurt"   = Lexer.True
convertToToken "eslaf"  = Lexer.False
convertToToken "loob"   = TBoolean
convertToToken "larutaN" = TNatural
convertToToken "lobmyS" = TSymbol
convertToToken "rahC"   = TChar
convertToToken "tsiL"   = TList
convertToToken t = ID $ reverse t

```

---

The `munch` function is finally used by `lexify`, which will continually munch the text until no text remains, producing the full list of munched tokens.

---

```

doLexify :: String → [Token]
doLexify [] = [EOF]
doLexify code = token : doLexify rest
  where (token, rest) = munch code

lexify :: String → [Token]
lexify code = BOF : doLexify code

```

---